IBM Informix® Dynamic Server™ (IDS)
IDS Problem Determination Tutorial Series

Database Engine Problem Determination: Critical Errors and Hangs

Table of Contents

Database Engine Problem Determination: Critical Errors and Hangs

About this tutorial

Introduction

The objective of this tutorial is to provide information on problem determination of critical errors and hangs in an IDS environment. The tutorial will provide you with guidelines for collecting appropriate information in the case of an engine crash, and will show how to use this data to isolate and/or reproduce the problem. You will also gain insight into various hangs and how to more accurately define a hang. Through examples, you will learn how to identify the source of a hang, collect useful diagnostic information and how to potentially alleviate a hung database. If the ultimate source of the hang cannot be identified, appropriate diagnostic information can be forwarded to technical support for further analysis.

To understand concepts in this tutorial, you should be familiar with basic IDS engine monitoring utilities and techniques.

Tutorial Conventions Used

When a tool or utility is first mentioned it will be shown in **bold** text.

All command statements and their output will be shown in a `monospaced` font.

Some examples will show specific command options which may change over time, which will always be documented in IDS documentation.

About the author

You can reach Jeff Laube by locating his e-mail address in the IBM Global Directory at http://www.ibm.com/contact/employees/us .

Section 1 IBM Informix Dynamic Server Errors

Section 1.1 Assertion Failures

On occasion, an internal problem within an IBM Informix Dynamic Server (IDS) might present itself as an assertion failure. An assertion failure message illustrates what type of failure has occurred, who is responsible for the failure, when it occurred and where pertinent information was collected. An assertion failure will not always cause the server to shutdown, but regardless of the outcome and the type of failure, diagnosing the problem is essentially the same.

Indication of the occurrence of an assertion failure will be noted in the server's message capture facility, typically referred to as the online.log. The location of the `online.log` is described in the variable, `MSGPATH`, which is located in the engine's `onconfig` file.

An assertion failure message in the `online.log` will have a format typically comparable to the following examples:

EXAMPLE 1:

```
10:19:03  Assert Failed: No Exception Handler
10:19:03  Informix Dynamic Server Version 9.30.UC1
10:19:03   Who: Session(2, informix@, 0, 0)
                Thread(8, sm_poll, 0, 1)
                File: mtex.c Line: 415
10:19:03   Results: Exception Caught. Type: MT_EX_OS, Context: mem
10:19:03   Action: Please notify Informix Technical Support.
10:19:03  stack trace for pid 16467 written to /tmp/af.3f0af77
10:19:03   See Also: /tmp/af.3f0af77, shmem.3f0af77.0
```

EXAMPLE 2:

```
09:13:04  Assert Failed: Memory block header corruption detected in
mt_shm_malloc_segid 7
09:13:04  Informix Dynamic Server Version 9.30.FC2
09:13:04   Who: Session(1599307, idsdb@panaka, 5581, 741958456)
                Thread(1718688, sqlexec, 12c3672e8, 1)
                File: mtshpool.c Line: 3206
09:13:04   Results: Unable to repair pool
09:13:04   Action: Please notify Informix Technical Support.
09:13:04  stack trace for pid 8 written to
/export/home/informix/log/af.3d885d0f
09:13:04   See Also: /export1/home/informix/log/af.3d885d0f,
shmem.3d885d0f.0
```

It may be the case that a database administrator may notice a sequence of assertion failures reported at roughly the same point in time. Typically, the primary focus should be directed to the initial or first reported assertion failure message. Often times, trailing

messages may be residual effects of the problem first identified by the initial assertion failure message.

Section 2 Evidence Collection

Section 2.1 Assertion Failure Files and Shared Memory Dumps

A main focus of trouble shooting assertion failures is to acquire diagnostic information particular to the problem and provide this information to IBM Informix technical support. The simplest method for acquiring an abundance of useful information relevant to the failure can be obtained via the evidence.sh script. The engine is made aware of the location of this script through the onconfig parameter, `SYSALARMPROGRAM`. It is important that the database administrator update this parameter to at least the provided template, $INFORMIXDIR/etc/evidence.sh.

This script is executed by the server at the point of an assertion failure. It is defined to provide insight, primarily through **onstat**, into the particulars of the engine at the moment of a failure. A database administrator may choose to add their own components to the evidence.sh file.

In addition to information about the failure that is reported directly by the server, the output of the evidence.sh is captured in what is commonly called an (af) file. The last line of the examples above illustrates the location and exact name of this capture file. The directory that will hold the assertion failure file is defined by the onconfig parameter, DUMPDIR.

It is very important for the database administrator to set the value of `DUMPDIR` to a directory in a file system that contains not only enough space for the contents of an af file, but also enough space to hold a much larger file, a shared memory dump. Especially when a particular failure is not reproducible, a shared memory dump is a very important file that can be created at the moment of a failure automatically by the database engine. A shared memory dump is essentially a file whose contents include all resident, virtual and communication memory segments used by the server. This snapshot of shared memory can be an invaluable source of information about an engine failure for a technical support engineer. Often times, the analysis of a shared memory dump can lead to the identification of a known bug or even uncover a previously unidentified problem in the engine. As previously mentioned, your `DUMPDIR` should be large enough to hold the contents of a complete shared memory dump. Its size can easily be determined by running `onstat -` against the running server. For instance, the following output from the `onstat -` command indicates a shared memory dump will be roughly 30 MB in size (30720 Kbytes).

```
Informix Dynamic Server Version 9.30.UC1     -- On-Line -- Up 00:00:46
– 30720 Kbytes
```

The capture of shared memory dumps at the point of a failure is enabled by setting the onconfig parameter, `DUMPSHMEM`, to 1 and can be turned off by setting this parameter to 0. The last line of both assertion failure messages noted above illustrate the location and file name of the shared memory dump in addition to the file name and location of the assertion failure file. So, in the first example of `Section 1.1`, the shared memory dump is located in the file `/tmp/shmem.3f0af77.0`.

Section 3 Interpreting the Assertion Failure Message

Section 3.1 Illustration

As in the second example of an assertion failure shown in `Section 1.1`, the failure may be particular to a user thread, namely in this case, an sqlexec thread as seen on line 4 of the message. Here we see the sqlexec thread with id, 1718688, was the thread that was running when the error occurred. The combination the this thread's id and also its corresponding session, 1599307, shown on line 3 of the message can give the database administrator insight as to what this user was doing at the time of the crash. Within the contents of the file, af.3d885d0f, will be various information about this session. The output of the command `onstat -g sql 1599307` can be located within this file and will show the database administrator the SQL statement that was current at the time of the failure and what database it was executed against. Some sample output follows:

```
==========------------ - - - - - -
/export1/home/informix/bin/onstat -g sql 1599307:

Informix Dynamic Server Version 9.30.FC2     -- On-Line -- Up 13 days
16:56:45 -- 1178624 Kbytes

Sess   SQL            Current            Iso Lock       SQL  ISAM F.E.
Id     Stmt type      Database           Lvl Mode       ERR  ERR  Vers
1599307 SELECT          tsmc_prod          CR  Wait 360   0    0    9.03

Current SQL statement :
   SELECT COUNT(*) FROM update_stats  WHERE error_code IS NULL and flag
= 2

Last parsed SQL statement :
   SELECT COUNT(*) FROM update_stats  WHERE error_code IS NULL and flag
= 2
```

Section 3.2 Goals

For several reasons, there is significance for determining the current SQL statement. A database administrator can execute this statement against their system to determine if the execution, alone, is enough to reproduce the problem in their server environment. Assuming a reproduction is available, steps can be made to reduce the reproduction to a more manageable test case outside of a potential production environment. In the case of engine failures, a test case is arguably the most important information that can be provided to a technical support engineer. Not only does a test case provide the means to

verify a problem's relevance to known bugs, but it also offers an invaluable means for debugging a heretofore unidentified issue. Another benefit from identifying an SQL statement that can cause an engine failure is avoidance. DBA's can alert application developers of offensive statements that can be removed from execution plans until a solution determined or a suitable workaround is uncovered.

When a reproduction is not available for a user session or, as in the case of the first example above, the failure is related to an engine thread like a poll thread, an important bit of information is obtained in the form of a stack trace. The default evidence.sh script will attempt to obtain the stack trace of the thread that caused the failure by running the command `onstat -g stk <tid>` where `<tid>` signifies this thread's id number. It is not uncommon for known bugs to be attributed to a given failure simply by noting similarities in reported stack traces.

In summary, trouble shooting assertion failures is best carried out by analysis of a potentially wide range of data. The most useful information can be obtained by the server's execution of the evidence.sh script. This diagnostic information coupled with a shared memory dump may offer great insight into the underlying source of a failure or it may provide valuable direction to a technical support engineer's efforts in pursuing a resolution. The most desirable information would be in the format of a test case and the output of the evidence.sh may offer the foundation of such a reproduction.

Section 4 IBM Informix Dynamic Server Hangs

Section 4.1 Connectivity Hangs

When trouble shooting hangs associated with IBM Informix Dynamic Servers, one of the first tasks should be focused on trying to define the hang. Often times, trying to narrow down the scope of the hang can help isolate the underlying problem. There are several questions a database administrator can pose to help with this task. Are only new connections hung or prevented from connecting to the database server? Are sessions associated with existing connections continuing to run or are there no running threads on the server? Is only an individual session hung or not returning after the execution of its last SQL statement?

Consider connectivity or network related hangs. It might be the case that only new connections are hung or not allowed to connect to the database server while existing threads can still process. It may also be that new connections are hung as well as existing connections. Things to consider in such circumstances include the current protocol, the current value of the `INFORMIXSERVER` environment variable and the various values for the onconfig parameter, `DBSERVERALIASES`. If connections using a value of `INFORMIXSERVER` that correspond to the socket protocol cannot connect to the database, can `INFORMIXSERVER` be changed to another value in the `DBSERVERALIASES` list that also is specific to the socket protocol and connect to the server? In other words, is there

something inherently wrong with the overall socket protocol in the database server or just with the components specific to a certain service of the socket protocol? If connections using a value of `INFORMIXSERVER` that correspond to one protocol (perhaps sockets) cannot connect to the database engine, can a connection be made if `INFORMIXSERVER` is changed to reflect a new protocol (perhaps shared memory or streams)?

If this classification of hang seems particular to a specific protocol, certain information regarding the respective poll threads and listen threads should be collected. The listen and poll threads play integral parts in accepting new connections. The listen thread is responsible for accepting a new connection and starting a session. The poll thread is responsible for telling the listen thread when new connection requests have arrived and when new messages for existing connections have arrived. If one of these threads encounters a problem, new connections attempts may be affected. It's important to try and understand what these threads are currently doing (or not doing) when this type of hang is encountered.

The first view into the current status of these threads can be obtained from the command `onstat -g ath`. This command lists all threads on the database engine. Note the poll and listen thread names for the various protocols:

```
protocol        poll thread             listen thread

TLI             tlitcppoll              tlitcplst
Sockets         soctcppoll              soctcplst
Streams         ipcstrpoll              ipcstrlst
Shared Memory   sm_poll                 sm_listen
```

Assume the TLI protocol is under consideration. The relevant output of a sample `onstat -g ath` follows:

```
Threads:
 tid     tcb      rstcb    prty status            vp-class     name
 9       b338d70  0        2    running             9tli     tlitcppoll
 12      b37a698  0        3    sleeping forever    1cpu     tlitcplst
 13      b308300  0        3    sleeping forever    3cpu     tlitcplst
```

This output represents typical status information for these threads. A poll thread is generally running and a listen thread is most commonly found sleeping forever. The poll thread is constantly checking for new incoming messages. If a message arrives and indicates a new connection is pending then the poll thread will wake the listen thread from its sleeping state to handle the connection request. If a hang of the network nature has been reported and one or all of these threads are consistently reported to be in some other state, then this may be an outward problem indication.

Section 4.2 Resource Hangs

When a hang is not related to connectivity, the main task is to try and isolate the source of the hang. It needs to be determined whether a single thread is waiting on some

particular item or possibly whether multiple threads are waiting on a common resource. An approach to identifying the resource a thread is waiting for can be initiated through a couple `onstat` commands, namely `onstat -u` and `onstat -g ath`.

An `onstat -u` lists all threads. When dealing with hangs, one column of significance in this `onstat's` output is the `wait` column. The `wait` column holds the address of a resource that this thread is currently waiting on. The address may be associated with items like locks, conditions, buffers or perhaps mutexes. A little legwork may be involved to determine what resource this is an address for, but a good starting spot would be to check the output of the commands `onstat -k`, `onstat -g con`, `onstat -b` and `onstat -g lmx` to check for locks, conditions, buffers and mutexes respectively. Another approach to identify the source of the address would be to search for the address within the output of an `onstat -a` or an `onstat -g all`. These `onstats` incorporate the output of many individual onstat options. The output below shows that the user `laube` is waiting on a resource having address, `0xb7982b8`.

```
Userthreads
address   flags   sessid  user   tty  wait     tout locks nreads nwrites
ae18818  Y--P--- 18      laube  14   b7982b8  0    1     0      0
```

A quick search through the output of `onstats` listed above finds the address in an `onstat -g con` output.

```
Conditions with waiters:
cid      addr     name               waiter   waittime
543      b7982b8  sm_read            40       43
```

The thread associated with session 18 is waiting on a condition named `sm_read`. This condition describes the fact that session 18 is associated with a front end that connected via shared memory and we are currently waiting for a new message from the front end.

An `onstat -g ath` can also be helpful for identifying what type of resource a thread is waiting on as well. For instance, session 18 above has a single user thread with thread id 40. The relevant portion of an `onstat -g ath` below shows that this thread has a status that would reinforce it is waiting on an `sm_read` condition.

```
Threads:
 tid     tcb      rstcb    prty status                vp-class    name
 40      b436bb0  ae18818  2    cond wait sm_read   1cpu        sqlexec
```

A hang may involve user threads waiting on a resource held by another user thread. Again, it will be helpful to determine specifics of the resource that threads are waiting on. What is the thread doing that holds the resource? An `onstat -g ses` on the session holding the resource may provide initial insight. Additional quality information may also be obtained by grabbing a stack trace for the thread holding the resource that is to be

discussed. Is the resource consistently held by the same thread or is it just a 'hot' resource desired by many threads?

Section 5 Evidence Collection

Section 5.1 What to Collect

Often, when hangs of any variety have been encountered, there will be considerable pressure on the database administrator of a production system to make the server accessible quickly. A common and speedy remedy is to bounce the server (bring it off-line and then back on-line). If the source of the hang is unidentified and there is no time for technical support to analyze the hung system, before bouncing the server as much information as possible should be collected to help with analysis after the fact. The following information is desirable:

1. Stack traces of relevant threads. If it's a network related hang the stack traces for poll and listen threads will be needed. For hangs not related to connectivity, a stack trace of the relevant sqlexec thread(s) may be necessary.
2. The output of the command `onstat -a`
3. The output of the command `onstat -g all`
4. A shared memory dump
5. For network related hangs, the output of the operating system command, `netstat -a`, may provide insight.
6. Relevant `onstat` output associated with the address of the held resource and session specific information for both the thread holding the resource and those threads that want the resource

Section 5.2 Stack Traces and Shared Memory Dumps

Consider this portion of an `onstat -g ath` output:

```
Threads:
 tid      tcb       rstcb     prty status            vp-class     name
 9        b338d70   0         2    running             9tli     tlitcppoll
 12       b37a698   0         3    sleeping forever    1cpu     tlitcplst
 13       b308300   0         3    sleeping forever    3cpu     tlitcplst
```

Regardless of the state, a stack trace for each relevant thread can be important information for technical support. To obtain a stack trace, first note the thread id (`tid`) in the `onstat` output above. When a thread has a status other than 'running', the easiest method for obtaining a stack trace is to run `onstat -g stk <tid>` where `<tid>` represents the thread id. Obtaining a stack from a running thread requires a different technique. In versions 9.21 and up of IDS, stack traces of running threads are obtained with the command **onmode** `-X stack <vpid>`. onmode -X stack command-line option doesn't seem to be documented The `<vpid>` component in this command can be obtained

from the vp-class column of the `onstat -g ath` output. Note the number prior to the class name. This represents the vpid. For instance, the tlitcppoll thread has an associated vp-class called '9tli'. The command to generate a stack for this running thread is `onmode -X stack 9`. A message comparable to the following will be present in the online.log after running this `onmode` command:

```
11:38:33  stack trace for pid 17327 written to /tmp/af.3f11399
```

The contents of this file contain the stack trace for the running thread.

In versions 7.x and those prior to 9.21 of IDS, stack traces for running threads are captured by the following command, 'kill -7 <vp_pid>'. The <vp_pid> component represents the process of the virtual processor on which this thread is running. Again, consider the above example output from `onstat -g ath`. The poll thread is running on the vp-class, `9tli`. You can map this virtual processor to a process id with the `onstat -g sch` command. The following output is taken from this `onstat` command:

```
VP Scheduler Statistics:
 vp    pid      class       semops    busy waits  spins/wait
 1     17319    cpu         31        42          9060
 2     17320    adm         0         0           0
 3     17321    cpu         3335      3341        9991
 4     17322    lio         3         0           0
 5     17323    pio         2         0           0
 6     17324    aio         24        0           0
 7     17325    msc         6         0           0
 8     17326    aio         21        0           0
 9     17327    tli         2         2           1000
 10    17328    aio         6         0           0
 11    17329    aio         5         0           0
 12    17330    aio         4         0           0
 13    17331    aio         4         0           0
```

The process id for the vp-class, '9tli' is 17327. So, on such versions, the command to generate a stack trace for this running poll thread would have been `kill -7 17327`. A message comparable to that above would have been written to the online.log and the stack trace can be found within the reported file.

The significance of shared memory dumps was noted in prior discussion of assertion failures and their capture was a result of mechanisms that are triggered by the engine at the time of failure. Shared memory dumps may also provide great insight for technical support into hung systems. For such systems, shared memory dumps will need to be captured manually. The command, `onstat -o <filename>`, can be executed to manually obtain a shared memory dump. The filename should be specified as the full path and file of the shared memory dump to be created. The file system that is used to hold the created shared memory dump should be large enough to accommodate the contents of a complete shared memory dump.

As there exist many varieties of engine hangs, diagnosing the different circumstance that may be particular to a hang is not an exact procedure. Different hangs warrant different diagnostic methods and the focus of the information in this article can provide much insight into the source of the hang and determining a resolution.

Summary

What you should know

You should now be familiar with methods and techniques for diagnosing errors and hangs that may occur on the IBM Informix Dynamic Server. Also, you should be familiar with information that can be collected to aid technical support should determining the underlying source of the problem require assistance.

For more information

For more information refer to Administrator's Guide for Informix Dynamic Server.